Неизбежность тонкого клиента

Алексей Фомкин
Тинькофф
https://fomkin.org

https://match3.fomkin.org

Тонкий клиент

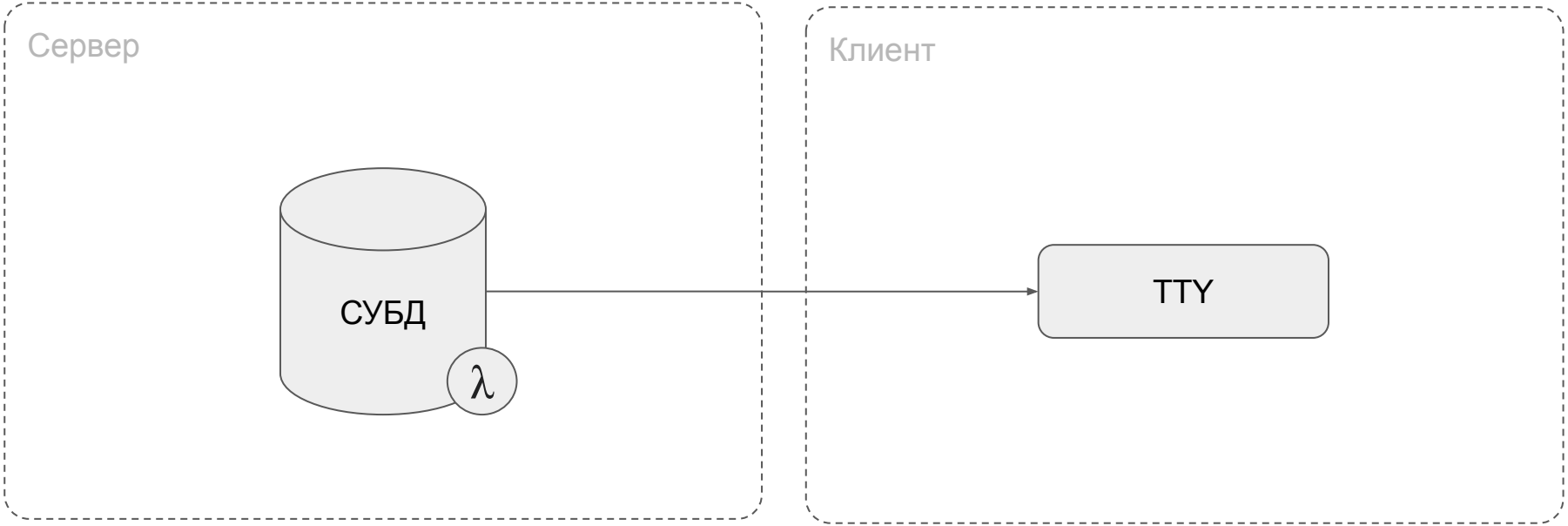
1. Состояние хранится на сервере
2. Логика выполняется на сервере
3. Клиент умеет отправлять ввод и принимать команды для осуществления вывода
4. Клиент ничего не знает по содержание приложения

Толстый клиент

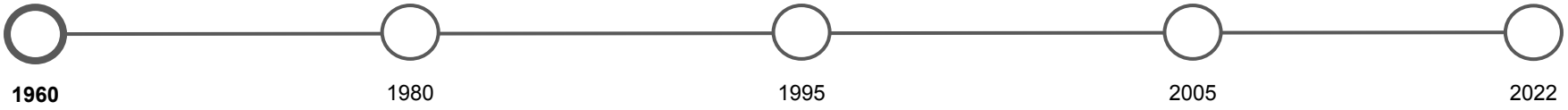
1. Состояние хранится на сервере и на клиенте
2. Логика выполняются на сервере и на клиенте
3. Клиент общается с сервером через API/RPC
4. При необходимости клиент может быть самостоятельным приложением

Эволюция приложений

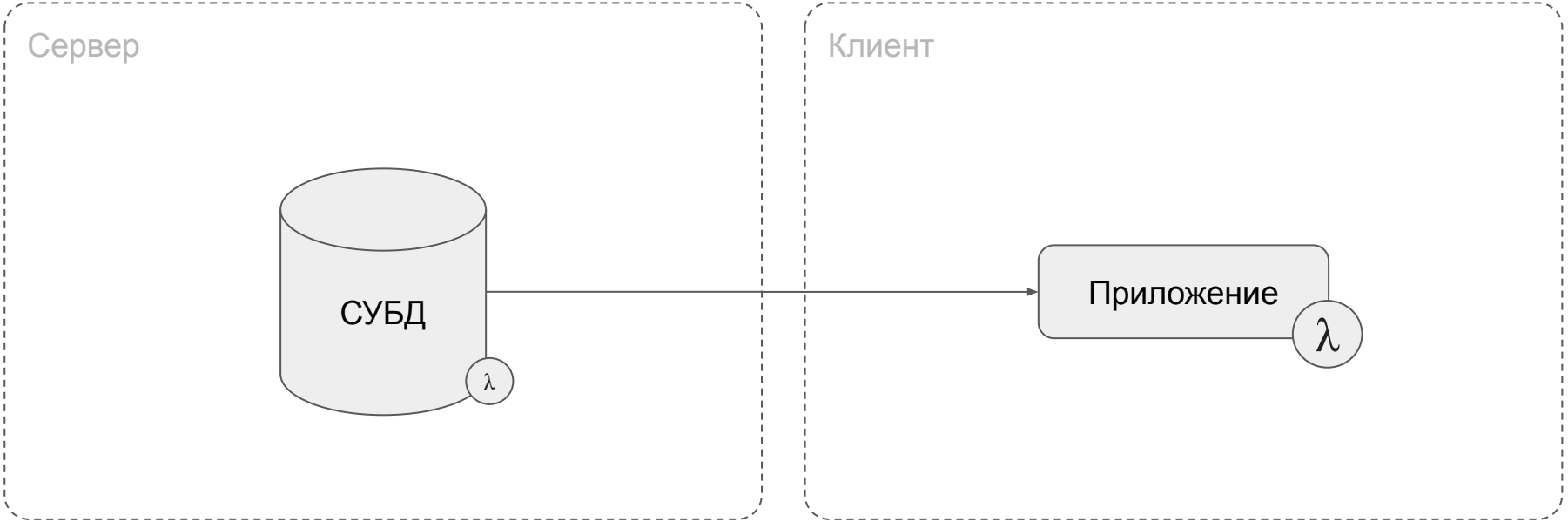
Эпоха терминалов



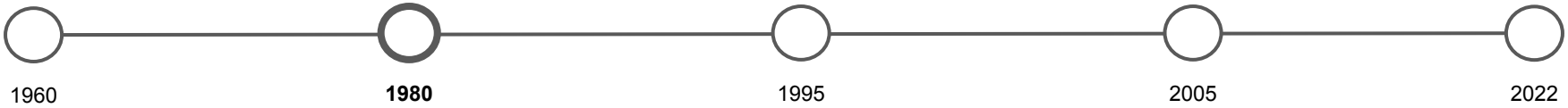
Данные и логика лежат вместе на одной машине, все работают через терминалы



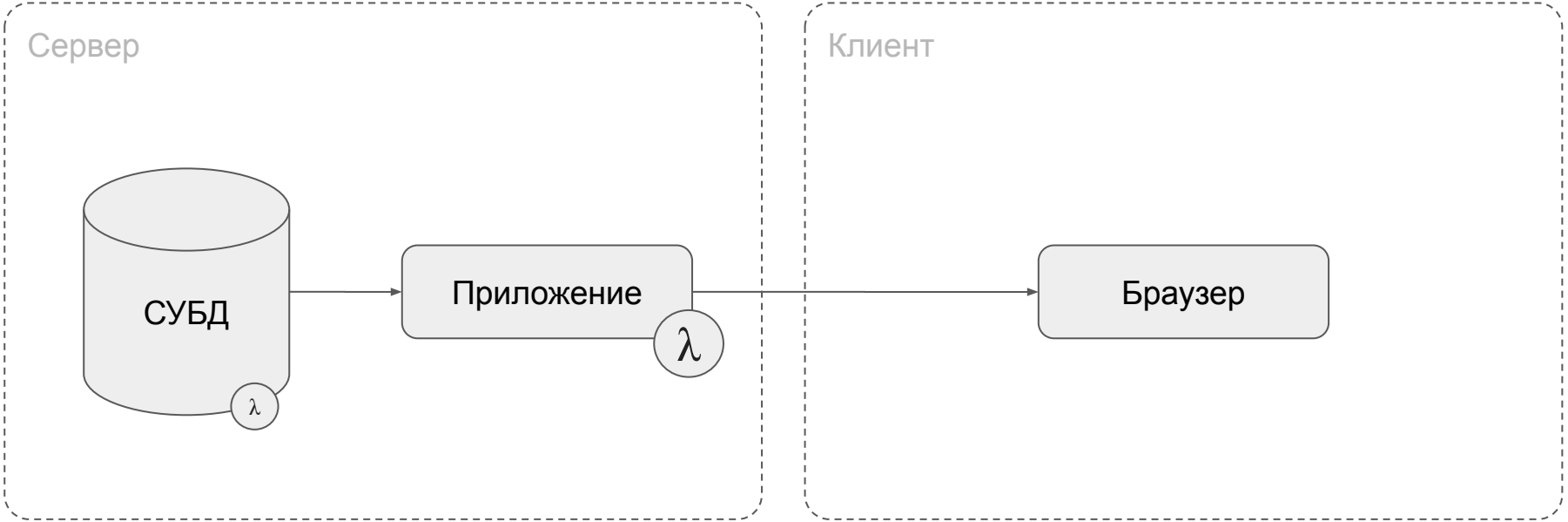
Эпоха ПК



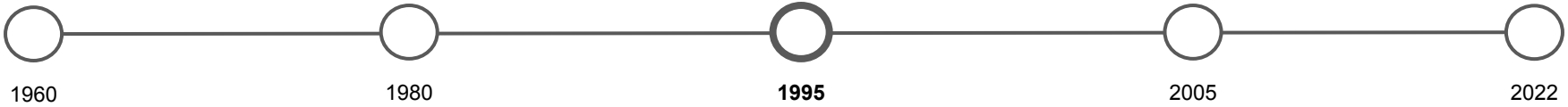
Данные лежат в СУБД, часть логики в хранимых процедурах, большая часть на стороне приложения.
Приложение подключается на прямую в СУБД.



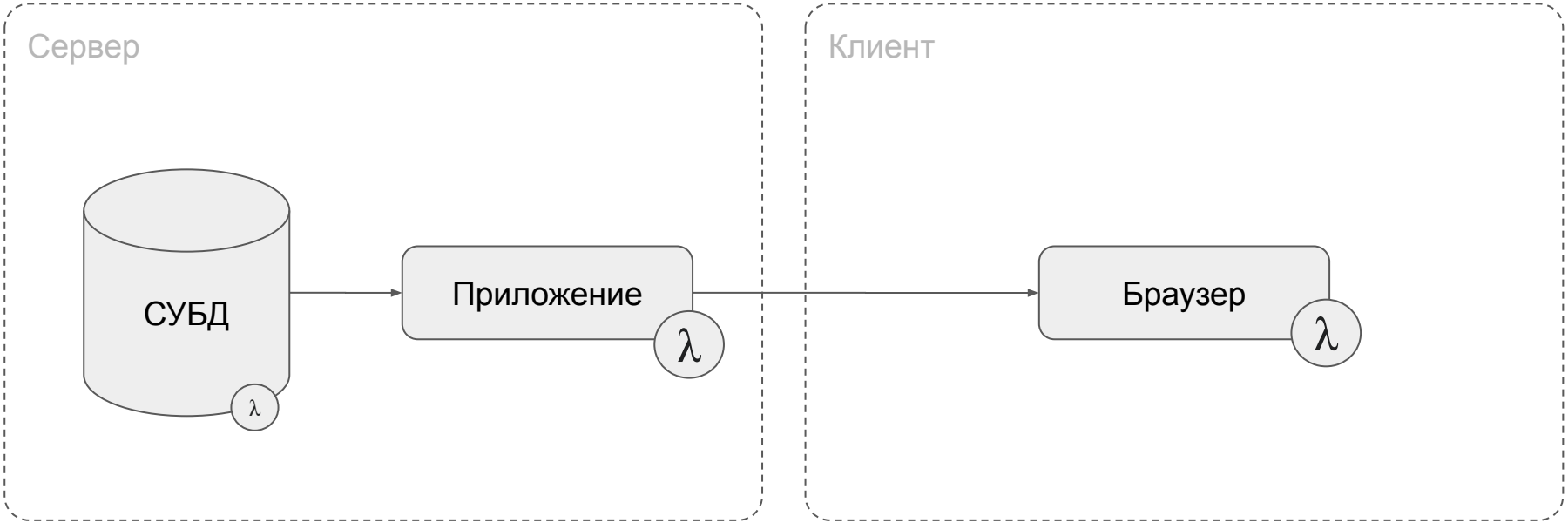
Становление интернета



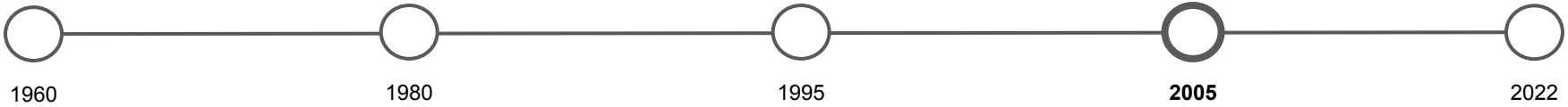
Данные лежат в СУБД, самая малая часть логики в хранимых процедурах, большая часть на стороне приложения. Ввод-вывод осуществляется через браузер.



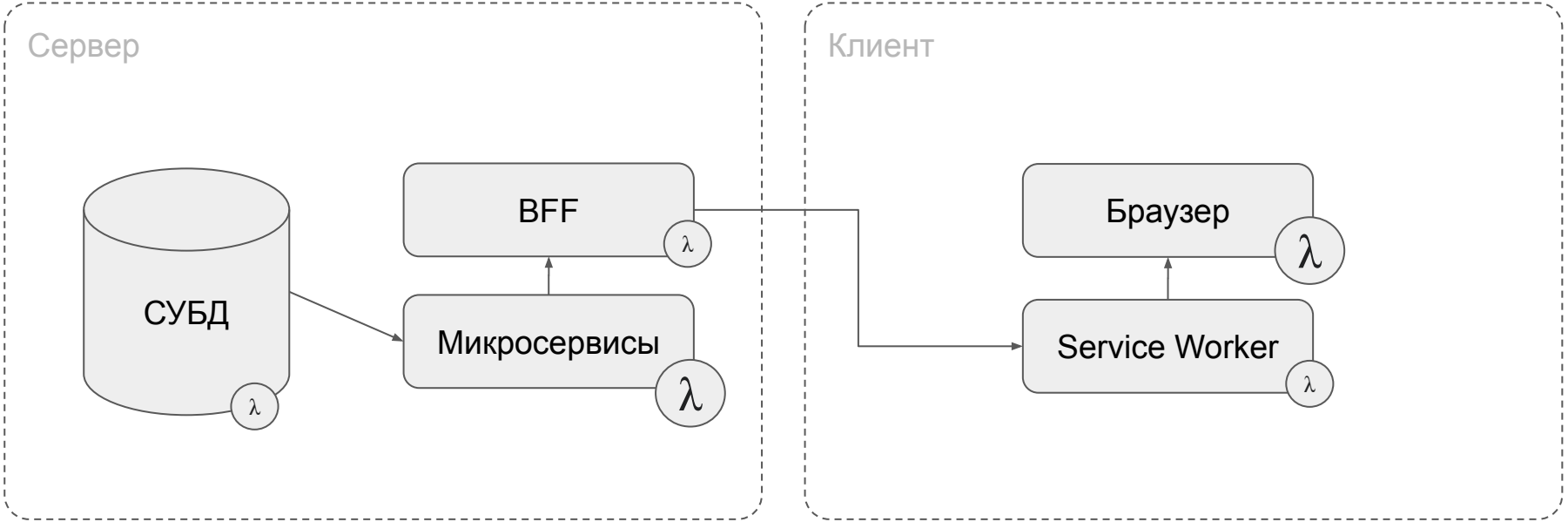
Эпоха JS-приложений



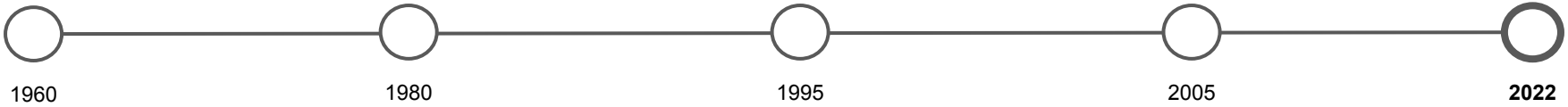
Данные лежат в СУБД, самая малая часть логики в хранимых процедурах, большая часть на стороне серверного приложения. Еще одно приложение работает на стороне клиента и работает с серверным по специализированному протоколу.



Современное состояние



Данные лежат в СУБД, большая часть логики на стороне серверного приложения. Роутинг на стороне BFF. BFF общается с бэком по RPC. BFF общается по RPC с service worker, который в свою очередь общается по RPC с браузером.



Общие позитивные последствия

1. Разделение труда (бэк, фронт)
2. Работа в оффлайне
3. Быстрый отклик

Общие негативные последствия

1. Логика размазана между клиентом и сервером
2. Логика массово дублируется
3. UX хуже (хотя хотели сделать лучше)
4. DX хуже
5. Больше ошибок (больше составных частей)
6. Нужно больше вычислительных мощностей и памяти
7. Нужно больше программистов
8. “Нужно больше золота”

Специальные негативные последствия

1. Большой размер клиента (>500 Кбайт)
2. Дублирование серверной логики
3. Высокое потребление CPU, RAM на клиенте
4. Высокое TTI (Time to Interactive)
5. Фризы на слабых клиентах
6. Высокие требования к квалификации разработчиков

Оправдана ли такая эволюция?

Может быть мы зашли куда-то не туда?

Можем ли упростить без потерь?

1. Разделение труда (бэк, фронт)
2. Работа в оффлайне
3. Быстрый отклик

1. Программист, верстальщик
2. Нужна ли она всегда?
3. Сети с низким latency: 4G, 5G, Starlink

Не просто можем, но и должны, потому что KISS.

Korolev

(в честь Сергея Павловича Королева)

Что это?

1. Библиотека для построения SPA на Scala
2. Развивается с 2016 года
3. 540 звездочек на Github (для Scala это не мало)
4. Используется в Зеленом Банке, Тинькофф, ряде стартапов
5. Три разработчика

На что похоже?

1. React (декларативное описание страницы)
2. Immutable.js (неизменяемые структуры данных)

В чем плюсы

1. Scala вместо JS/TS
2. Server-side rendering “из коробки”
3. Роутинг из коробке
4. Server-push из “коробки”
5. Не нужно писать клиент-серверный API (подключение БД и микросервисам напрямую)
6. Низкое потребление памяти и CPU на стороне клиента
7. Маленький размер JS-кода приходящего на клиент (~10 Кбайт без сжатия)
8. Мониторинг на стороне сервера, большая повторяемость
9. Конечная стоимость разработки ниже

В чем минусы

1. Scala вместо JS/TS
2. Не работает в оффлайне вообще
3. Чувствителен к latency в сети (выше 100 миллисекунд)
4. Анимации только CSS
5. Дополнительная память на стороне сервера
6. Все считают тебя фриком

Почему вам это нужно?

1. Хочу чтобы страница грузилась мгновенно
2. Хочу чтобы не тормозило (в том числе на старом железе)
3. Хочу делать огромные системы с 1000 форм
4. Хочу разрабатывать быстро, не париться с API
5. Не хочу общаться с душными бэкендерами

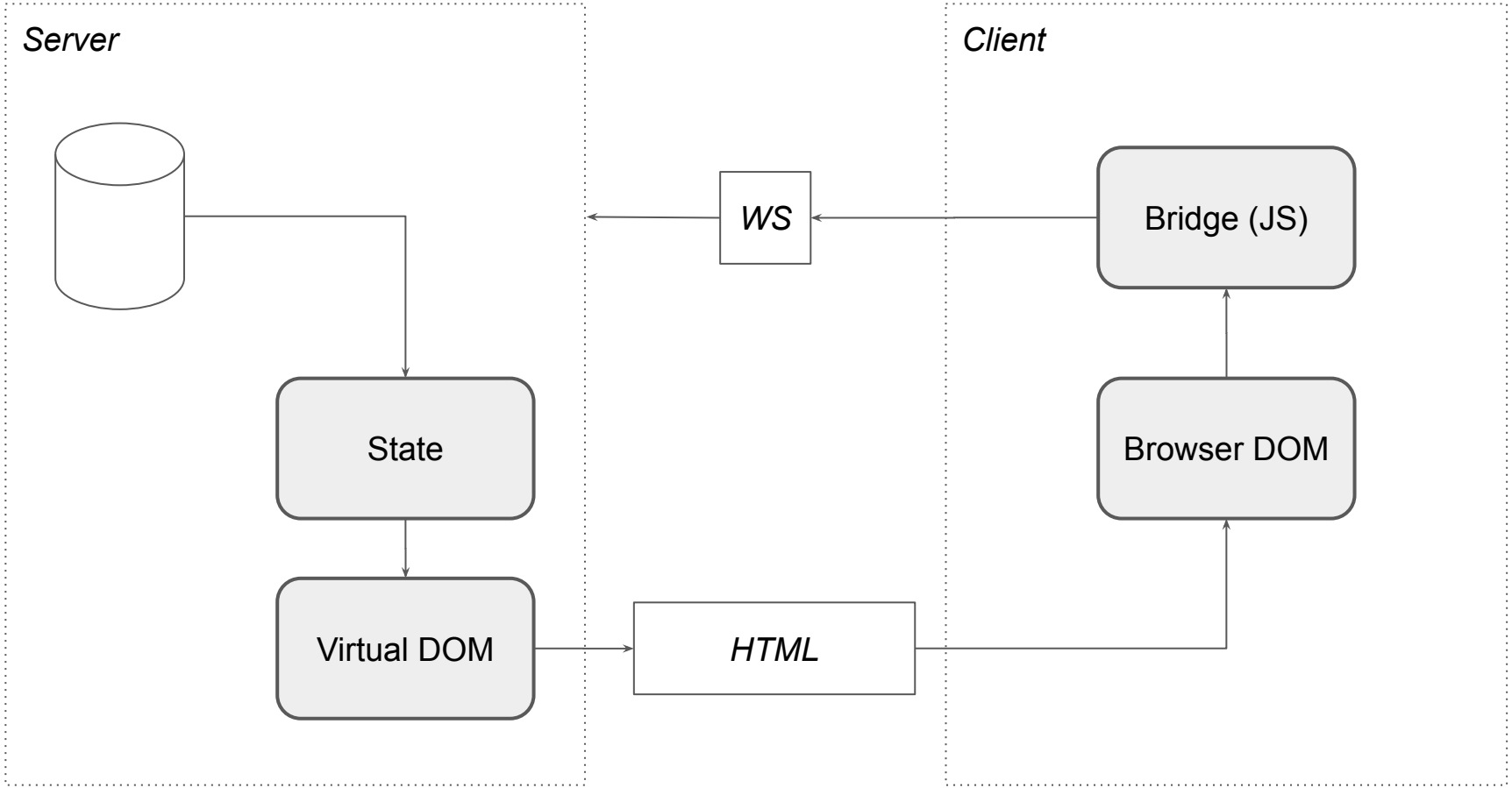
Рассмотрим пример на React

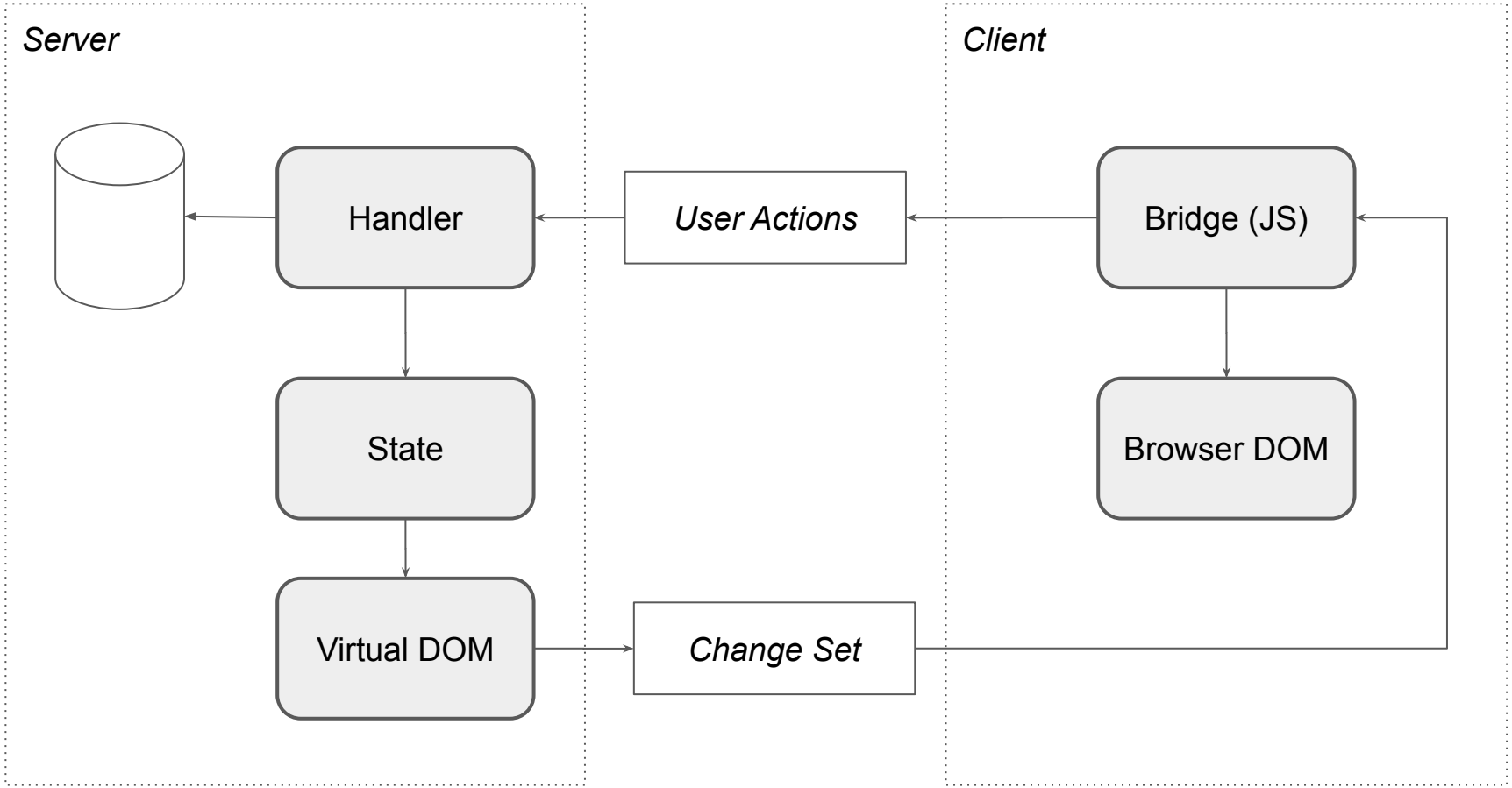
```
const copy = (o, field, value) => Object.assign({}, o, { [field]: value} );

function app(state, container) {
  function valueOfInput(event) {
    app(copy(state, 'input', event.target.value),  container)
  }
  function addItem(event) {
    event.preventDefault();
    app(copy(state, 'items', state.items.concat([state.input])),  container)
  }
  ReactDOM.render(
    <div>
      <div className="title">{state.title}</div>
      <ul>{state.items.map((item) => <li>{item}</li>)}</ul>
      <form onSubmit={addItem}>
        <input onChange={valueOfInput} />
        <button>Add</button>
      </form>
    </div>,
    container
  )
}
```

То же самое на Korolev

```
case class State(title: String, items: Vector[String])

val newItemInput = ElementId("newItemInput")
val render = (state: State) => body(
 div(clazz := "title", state.title),
 ul(clazz := "list", state.items.map(item => li(clazz := "item", item))),
 form(
   input(newItemInput),
   button("Add"),
   event("submit") { access =>
     for {
       newItem <- access.valueOf(newItemInput)
         <- access.transition(state => state.copy(items = state.items :+ newItem))
     } yield ()
   }
 )
)
```

Как это работает?


Server
Client
WS
Bridge (JS)
State
Browser DOM
Virtual DOM
HTML


Server
Client
Handler
User Actions
Bridge (JS)
State
Browser DOM
Virtual DOM
Change Set

И что, JS тут совсем нет и он не нужен?

Нужен!
Куда же мы без JS

https://fomkin.org/korolev/user-guide.html#_javascript_interoperability

Где пригодится JS

1. Продвинутые фичи (криптография, растровая графика, GPU-ускоренная графика)
2. Что-то требующее быстрой реакции на ввод
3. Анимации

Прямая коммуникация браузером

1. Можно выполнить произвольный код в браузере со стороны сервера. Поддерживаются Promise.
2. Можно зарегистрировать колбэк и вызвать сервер со стороны клиента.

Из сервера

```
form(
  input(newItemInput),
  button("Add"),
  event("submit") { access =>
    access.evalJs(js"$newItemInput.focus()")
  }
)
```

Из клиента

```
// Scala
access.registerCallback("myCallback") { myArg =>
 Future(println(myArg))
}

// JS
Korolev.invokeCallback('myCallback', 'myArgValue');
```

Custom Elements

1. CE это тот же самый DOM. Для Королева не разницы с чем работать с HTML или с CE.
2. Изолированное состояние
3. Хорошая поддержка в браузерах (94%)
4. Можно писать приложение на CE и использовать Королев как клей.

Custom Elements

```
val leafletMap = TagDef("leaflet-map")
val latitude = AttrDef("latitude")
val longitude = AttrDef("longitude")
val zoom = AttrDef("zoom")

body(
 leafletMap (
   width @= "500px", height @= "300px",
   latitude := state.lat.toString,
   longitude := state.lon.toString,
   zoom := "10"
 )
)
```

Попробуем?

Выводы

1. Королев позволяет сделать приложения проще и качественнее
2. Позволяет уменьшить затраты на создание приложений
3. Позволяет оставить UX на высоком уровне комбинируя подходы

Призыв к действию!

1. Попробуйте Korolev, он крутой
2. Изучите Scala, она еще круче
3. Приходите работать в Тинькофф, у нас просто отлично!

Спасибо за внимание!
Вопросы?

С вами был Леша Фомкин.

- a@fomkin.org
- https://github.com/fomkin/korolev
- https://fomkin.org